SMS: A Directive-Based Parallelization Approach for Shared and Distributed Memory High Performance Computers

M. W. Govett, D. S. Schaffer[1], T. Henderson, L.B. Hart, J. P. Edwards[2], C.S. Liou[3], T-L. Lee

Advanced Computing Branch
Aviation Division
NOAA Forecast System Laboratory
325 Broadway
Boulder, Colorado 80303

11/15/99

---

[1] Cooperative Institute for Research in the Atmosphere (CIRA)
[2] Instituto Nacional de Meteorologia,Brasilia, Brazil

[3] Naval Research Laboratory, Monterey, CA

Abstract

A directive-based parallelization tool called the Scalable Modeling System (SMS) is described. The user inserts directives in the form of comments into existing FORTRAN 77 code. SMS translates the code and directives into a parallel version that runs efficiently on both shared and distributed memory high-performance computing platforms. While the tool has been tailored toward finite difference approximation and spectral atmospheric models, the approach is sufficiently general to be applied to other structured grid codes. Results from two case studies suggest that the parallel approach scales well. However, further testing is required.

Keywords: Parallelization; directive-based; structured grids; atmospheric model

1. Introduction

Atmospheric modelers have become increasingly dependent on high performance parallel computers to meet their needs. To efficiently use these computer systems modelers must address issues such as portability, programmability, and performance. Portable performance, the ability for a code to run efficiently on multiple architectures, is important for two reasons. First, computers quickly become obsolete; typically a new generation are introduced every two to four years. Second, climate and weather modelers frequently share their codes with others who utilize different computing platforms. The primary mission of the National Oceanic and Atmospheric Administration's (NOAA) Forecast Systems Laboratory (FSL) is to transfer atmospheric science technologies to operational agencies (such as the National Weather Service) both inside and outside of NOAA. Thus, the most important requirement for FSL parallelization efforts is that the resulting code be able to run efficiently on a variety of architectures.

Typically, atmospheric models have been under development for years. They are often derived from collaborations between multiple scientists and institutions. In addition, the modelers who maintain the codes generally prefer to concentrate on the scientific aspects, avoiding computational issues such as vectorization and parallelization. This combination of factors imposes two additional requirements on any parallelization effort: modifications to serial code and the effort to parallelize should be minimized.

In the past decade, several distinct approaches have been offered to meet one or more of these requirements. One class of solutions is a directive-based micro-tasking (loop level) approach offered by companies such as Cray. More recently, OpenMP, a standard for such a set of directives, has become accepted in the community. OpenMP can be used to quickly produce parallel code, with minimal impact on the serial version. Unfortunately, the approach does not work for distributed memory architectures.

A message passing library such as Message Passing Interface (MPI), represents a second class of solutions suitable for shared or distributed memory architectures. The scalability of parallel codes using these libraries can be quite good, although likely lower than the micro-tasking class for small shared memory machines. The MPI libraries are relatively low-level. This requires the modeler to expend significantly more effort to analyze dependencies and parallelize their code. Further, the resulting code may differ substantially from the original serial version.

A third class of solutions is the parallelizing compiler. This class offers the promise of automatically producing a parallel code that is portable to shared and distributed memory machines. The compiler does the dependence analysis and offers the user directives and/or language extensions that reduce the development time and the impact on the serial code. The most notable example of a parallelizing compiler is High Performance FORTRAN (HPF). In some cases the resulting parallel code is quite efficient (The Portland Group, 1999). However, there are also deficiencies in this approach. Compilers are often forced to make conservative assumptions about data dependence relationships, which slow down the code (Ierotheou, et al., 1996). In addition, weak compiler implementations by some vendors result in widely varying performance across different systems (Frumkin, et al., 1998; Ngo, et al., 1997).

A fourth class of solutions is interactive parallelization tools. One such tool, called the Parallelization Agent, automates the tedious and time consuming tasks while requiring the user to provide the high level algorithmic details (Kothari and Kim, 1997). Another tool, called the Computer-Aided Parallelization Tool (CAPTools), attempts a comprehensive dependence analysis (Ierotheou, et al., 1996). This tool is highly interactive, querying the user for both high level information (decomposition strategy) and lower level details such as loop dependencies and ranges that variables can take. Both of these tools offer the possibility of a quality parallel solution in a fraction of the time required to analyze dependencies and generate code by hand. However, although the generated code is recognizable, the sequential and parallel versions of the source are distinctly different.

A fifth class of solutions are library-based tools such as RSL (Michalakes, 1994), and the Scalable Modelling System (SMS) (Rodriguez, et al., 1996). These tools are built on top of the lower level libraries and serve to relieve the programmer of handling many of the details of message passing programming. Performance optimizations can be added to these libraries that target specific machine architectures. Unlike computer-aided parallelization tools such as CAPTools however, the user is still required to do all dependence analysis by hand.

In simplifying the parallel code, these high level libraries also reduce the impact to the original serial version. Parallelization is still time consuming and invasive however, since code must be inserted by hand and multiple versions must be maintained. To further reduce this impact, source translation tools have been developed to help modify these codes automatically. One such tool, the Fortran Loop and Index Converter (FLIC), generates calls to the RSL library based on command line arguments that identify decomposed loops needing parallelization (Michalakes, 1997). While useful, this tool has limited capabilities.

Here, we present a directive-based translation tool that is a new addition to SMS called the Parallel Pre-Processor (PPP). The programmer inserts the directives (as comments) directly into the FORTRAN 77 serial code. PPP then translates the directives and serial code into a parallel version that runs on shared and distributed memory machines. Since the programmer adds only comments to the code, there is little impact on the serial version. Further, SMS hides enough of the details of parallelism to significantly reduce the coding and testing time compared to an MPI based solution. While SMS has only been applied to atmospheric model codes, the approach is sufficiently general to work for other structured grid models.

The remainder of this paper explains the details of how SMS works. Section 2 summarizes important parallel issues encountered in typical model codes. Sections 3 and 4 describe the structure of SMS and explain how the directives help the programmer resolve these parallel coding issues. Section 5 presents two case studies and offers performance results. Finally, section 6 concludes and suggests future enhancements to the tool.

2. Parallel Coding Issues

In any parallelization effort, the most important issue is to determine how to divide the work among the processors. In order to enable the problem to scale to large numbers of processors, the most common approach for the structured grids found in atmospheric model codes is to use a data domain decomposition. In this case, parallelism is usually achieved by having multiple processes execute the same program on different segments of the distributed data. This type of parallelism is called Single Program Multiple Data (SPMD).

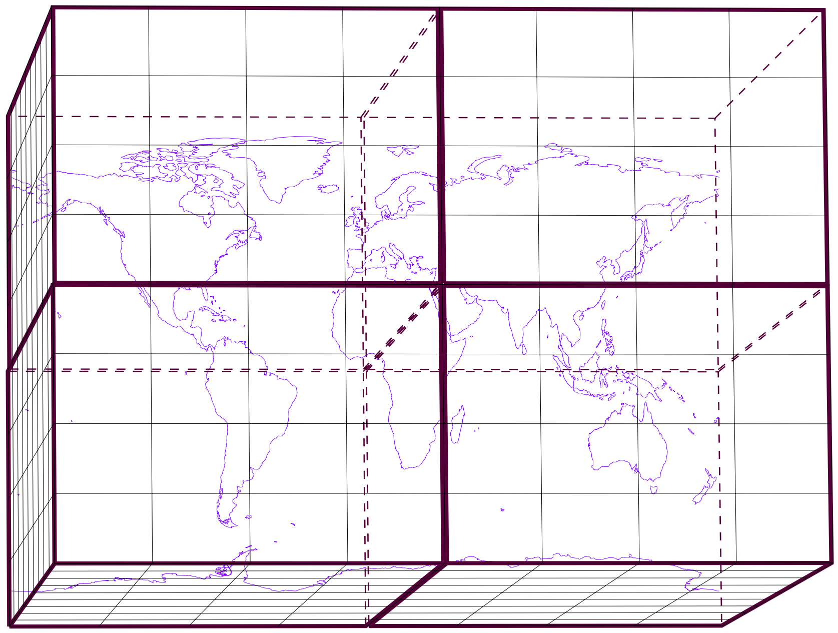


Figure 1: Representation of a horizontal data domain decomposition. Thin lines demarcate model grid boxes; thick lines indicate processor boundaries. In this case, the model data are divided among 4 processors.

In Figure 1, a horizontal data domain decomposition is shown where each processor controls the data in a slab from the surface to the top of the atmosphere. This decomposition is often used in finite difference approximation (FDA) weather models since complex dependencies for the physical parameterizations are rarely encountered in the horizontal.

When a data domain decomposition is chosen, four additional issues must be resolved in any parallelization effort. First, the code must be analyzed to determine where data dependencies occur. For example, the computation of y(i,j) in the statement:

$$y(i,j) = x(i+1,j) + x(i-1,j) + x(i,j+1) + x(i,j-1)$$

depends on the i+1, i-1, j+1, and j-1 points of the "x" array. This is called an adjacent dependence. When these data points are not local to the processor, they will need to be obtained from another processor. It is critical to understand all data dependencies in the source code. Further information about dependence analysis can be found in references on parallel programming such as Hwang (1993).

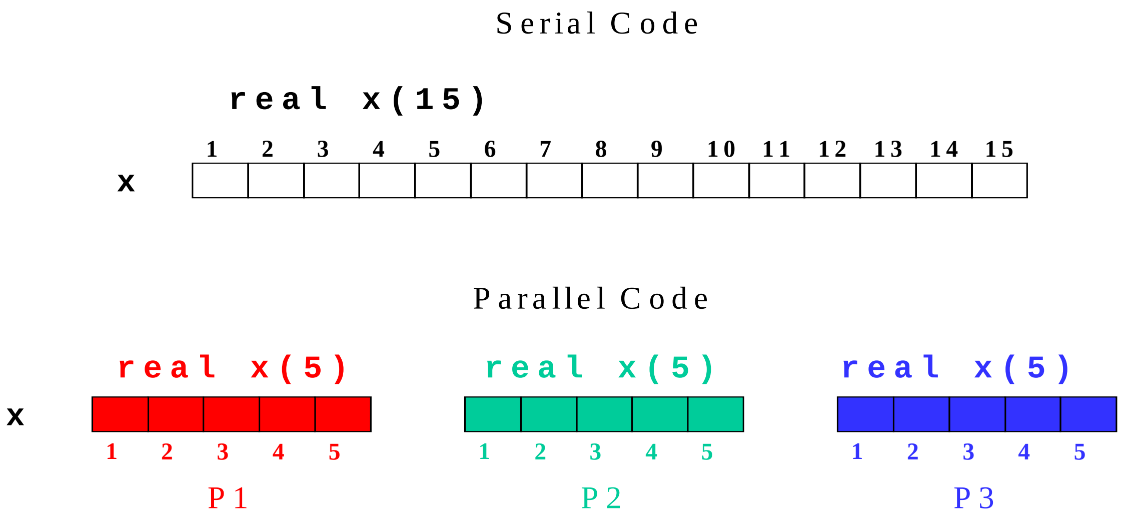


Figure 2: Schematic of array declarations for serial code and the corresponding 3 processor parallel version. In this case, the number of processors divide the sequential array size evenly.

Once data dependencies are understood, several critical coding problems must be handled. First, the FORTRAN 77 loops must be modified to operate only on the data local to each processor. Second, in order for the memory usage to remain roughly constant with an increasing number of nodes, the array declarations must be modified to cover only the processor local domain (Figure 2). Third, some mechanism may be required to make conversions between global and local array indices. For example, if decomposed arrays are indexed locally, the handling of model boundary conditions will require a conversion of an array index to its global equivalent (Figure 3).

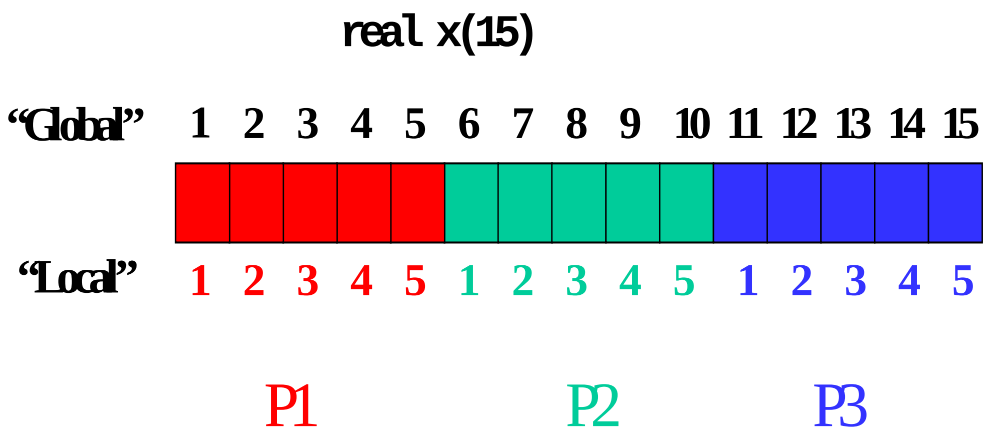

real x(15)
"Global" 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
"Local" 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
P1
P2
P3

Figure 3: Schematic showing the relationship between global and local array indices for the case when an array is decomposed among 3 processors.

A second critical issue involves how to handle the communication needed to resolve data dependencies. Since each processor contains a sub-domain, access to data on other processors may be required. The most common solution to handle adjacent dependencies uses halo or ghost regions (Figure 4). Each processor sends the edges of its data to its neighbors where it is stored in the halo regions. Then, loop calculations can be executed completely over each processor's local domain.

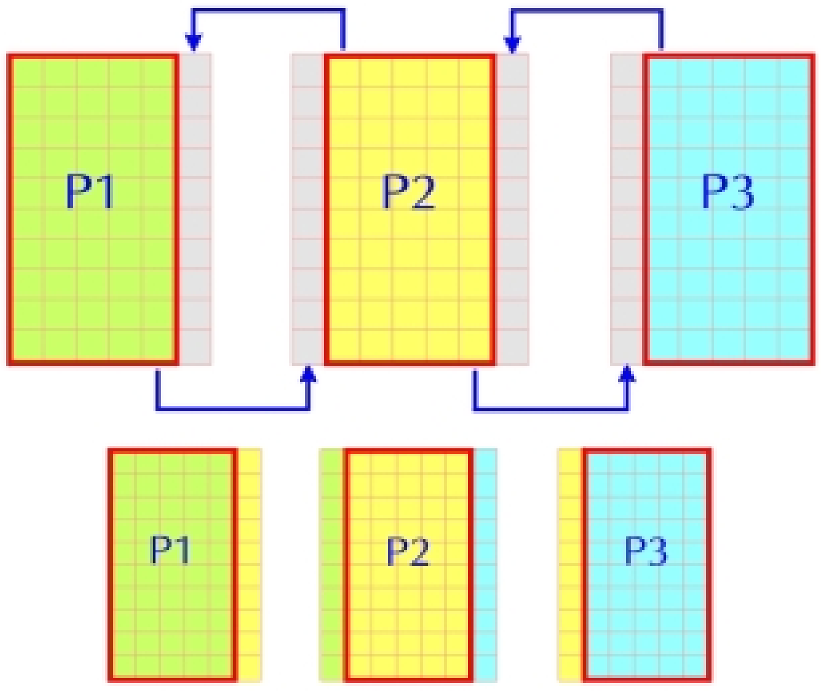


Figure 4: Schematic of how communications are implemented to handle adjacent dependencies using halo regions and "exchanges". The first column of P2 is sent to P1 where it is stored in the halo region just to the right of P1's data. The last column of P1 is stored in the left halo region of P2. The other communication works analogously.

Another important dependence, called a global dependence, occurs when all points in a given decomposed dimension are used in calculations such as summation. This requires communication among all processors in that dimension (Figure 5). For real and complex summations, re-ordering these sums can cause round-off errors. Since atmospheric models simulate non-linear systems, such errors can result in completely different solutions within a matter of a few model days. Therefore, for testing purposes, it is desirable to provide a mechanism for cross-processor summations that yield the bit-wise exact same answer regardless
Of the number of processors applied to the code.

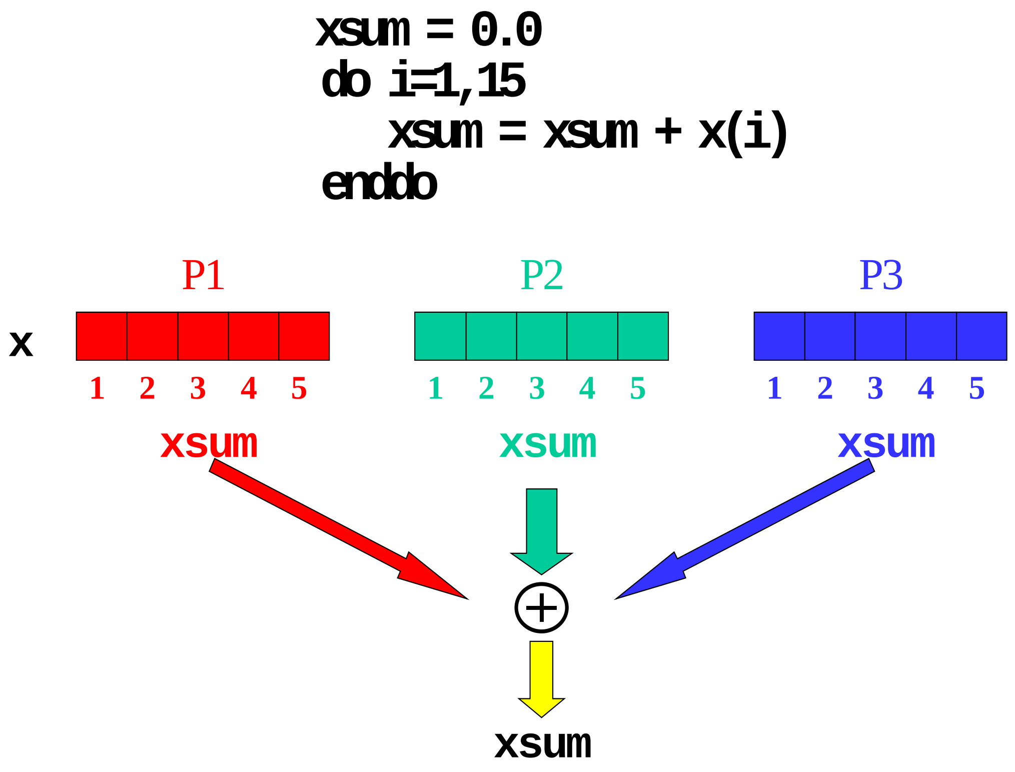


Figure 5: Schematic of a global summation. Each processor computes its local sum; storing it in “xsum”. The local “xsum” values for all processors are then added to produce the global sum of the decomposed array “x”.

A third critical issue is how to handle input and output of decomposed variables. Typically on input, one processor reads the data and scatters it appropriately to the others (Figure 6). On output, a gather of the decomposed data is followed by a write by a single processor. Alternatively, each processor could write their local data to disk; a post-processor would then be required to reassemble the decomposed arrays. Another alternative supported on some systems is parallel I/O, where multiple processors write to disk.

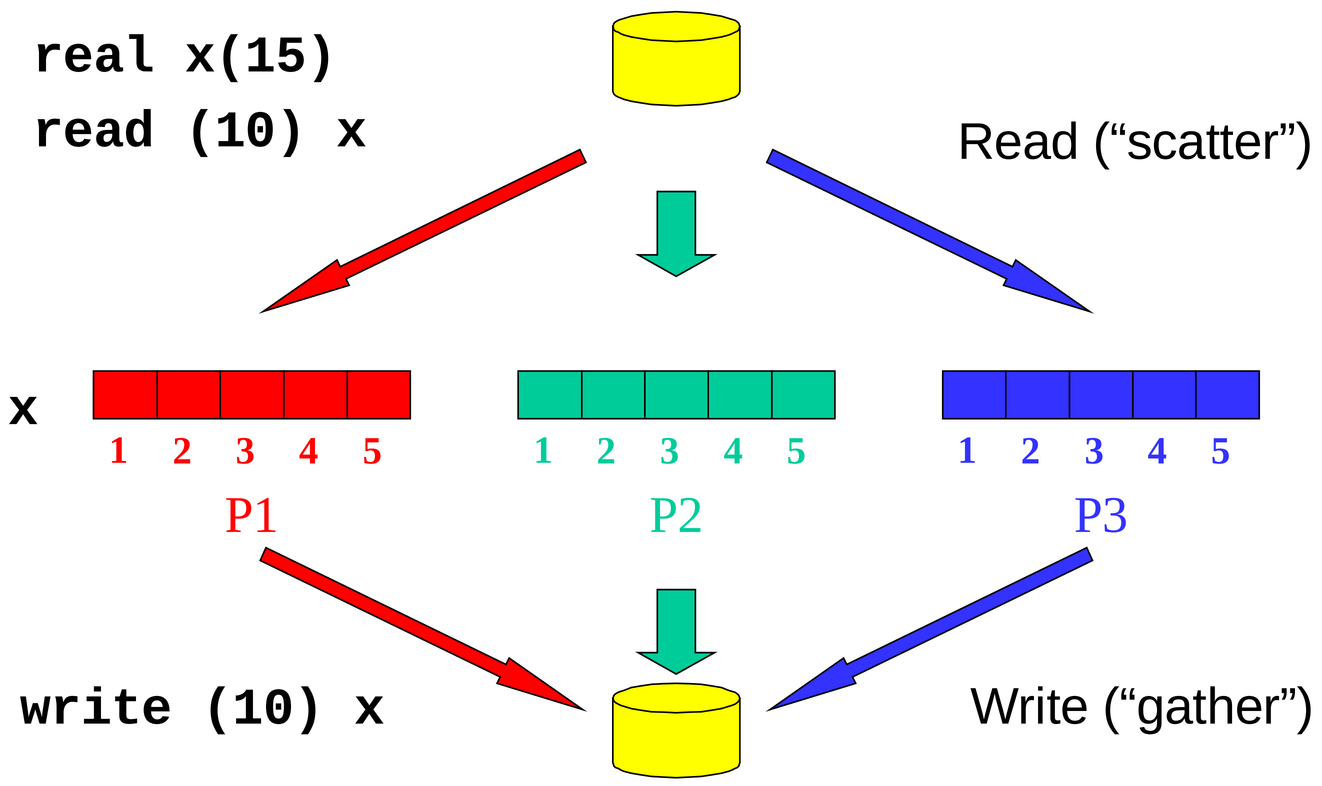


Figure 6: Schematic of the input and output of a decomposed array. On input, one processor reads the global data from disk. The appropriate sections of the global array are then "scattered" to each processor. On output, the decomposed data are gathered into a global array and then written to disk by one processor.

The fourth critical issue is how to transform data between different domain decompositions. In some models, computations may occur in alternating phases with each phase having different dependencies. One solution is to create a decomposition for each phase. In nested models, for example, fine and coarse will require different decompositions. Alternatively, in the spectral atmospheric code discussed in Section 5 (for 1-dimensional decompositions), a decomposition in latitudes is optimal for the physics and Fast Fourier Transform code while a decomposition in longitudes or in the vertical is ideal for the Legendre transforms.

Once these critical issues are addressed, the programmer can apply advanced techniques to improve performance. Load balancing strategies are important in optimizing performance. Both static and dynamic load imbalances should be considered. One example of a static load imbalance in a global atmospheric model occurs in short wave radiation calculations. These calculations are only needed in the daytime; processors whose domain contains nighttime points will have less work. The effect of this load imbalance becomes more pronounced as the number of processors increases. To see this, imagine each processor has only two grid points. If one of these processors has two night points and another has two day points, then the short wave calculations are running at only 50% efficiency. Performance will improve by re-distributing day and night points among the processors. One example of a dynamic load imbalance in atmospheric models occurs in convection calculations. Clouds can exist for a period of time in some regions and not others. Processors containing these convective regions will be busy doing extra calculations while the other processors will be idle. As the clouds move or change over time, the load imbalance shifts.

Performance trade-offs relating to the target system architecture should be considered in areas including the processor type, memory organization and inter-processor communications speed. For example, vector processors will perform better with long vector lengths while cache-based machines will do better with short ones. Inter-processor communications, a combination of the speed in which messages can be moved between processors (bandwidth) and the time required to set up these communications (latency), can vary widely between architectures. There are several common techniques used to reduce communication costs. On parallel machines that support asynchronous communications, messages can be transferred at the same time as model calculations are being done. In addition, combining messages into a single long message (aggregation) will reduce communications latency. Another strategy allowed by SMS and explained in Section 3, trades communications for redundant computations in the halo regions.

3. Overview of SMS

SMS was designed to support SPMD parallelism on both shared and distributed memory systems. To ensure portability across these systems, SMS assumes memory is distributed; no processor can address memory belonging to another processor. A local address space is used to access data by each processor; SMS provides mechanisms to translate global addresses into processor local references. Communications between processes are implemented using the message passing paradigm. Despite this assumption, the performance on shared memory architectures is good due to efficient implementations of message passing on these systems.

SMS consists of a layered set of four components (Figure 7). PPP, the highest layer, is a Fortran code analysis and translation tool built using the Eli compiler construction software (Gray, et al., 1992). PPP analysis ensures consistency between the serial code and the user inserted SMS parallelization directives. After analysis, PPP translates the directives and serial code into a parallel version. In addition to loop translations, array re-declarations, and other code modifications, PPP generates calls to routines contained in the Nearest Neighbor Tool (NNT), Scalable Spectral Tool (SST) and Scalable Runtime System (SRS) shown in the Figure 7. NNT is a set of high-level library routines that help address parallel coding issues such as data decomposition, halo region updates and loop translations (Rodriguez, 1996). SRS provides support for input and output of decomposed data (Hart, et al., 1995). SST is a set of library routines that support parallelization of spectral atmospheric models. These libraries rely on MPI routines to implement the lowest layered functionality required.

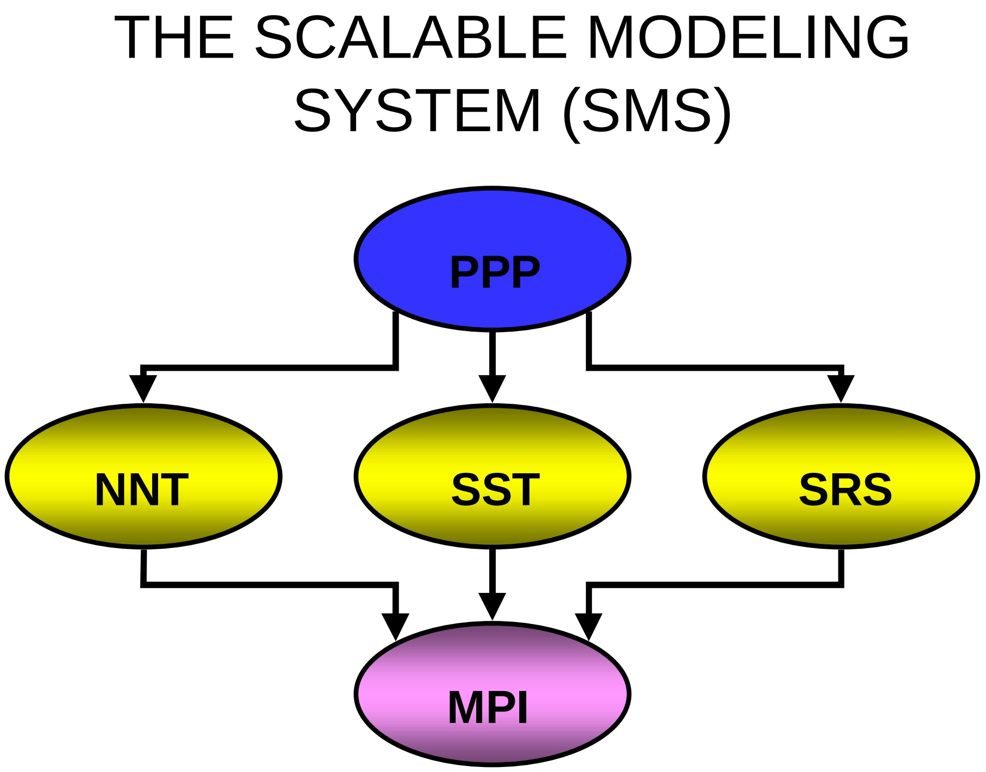


Figure 7: SMS is composed of a layered set of components. The modeler inserts comment-based directives into the serial code. These comments and serial code are translated into a parallel version that includes calls to the NNT, SRS and SST libraries. These libraries depend on the underlying message passing library, currently MPI.

SMS provides several techniques for optimizing performance. One is to make platform specific optimizations in the message passing layer. During a recent FSL procurement, one vendor replaced the MPI implementation of key SMS routines with the vendor's native communications package to improve performance. No changes to the model codes were necessary.

SMS also supports performance optimizations in inter-processor communications. One strategy is to trade communications for redundant calculations. Under this scheme, each processor does extra computations in the halo regions to eliminate some data exchanges with its neighbors. Figures 8a,b shows how this works. In Figure 8a, exchanges are required after loops 150 and 250. However, in 8b, the loops are executed one point into the halo region in each direction; eliminating the need for the exchanges after loop 150. On machines with low communication costs it may be better to avoid these redundant computes and simply exchange data after each loop. SMS also supports aggregation when exchanging multiple model variables in order to reduce communications latency.

```
C Assume entire halos of a and b valid before loop 150
C Assume entire halos of x must be valid after 250
      do 150 i=LOCAL_START_INTERIOR,LOCAL_END_INTERIOR
         y(i) = a(i) - a(i+1) - a(i-1)
         z(i) = b(i) - b(i+1) - b(i-1)
  150 continue
EXCHANGE y and z HERE
      do 250 i=LOCAL_START_INTERIOR,LOCAL_END_INTERIOR
         x(i) = y(i)*z(i) + y(i+1)*z(i-1)
              + y(i-1)*z(i+1)
  250 continue
EXCHANGE x HERE
```

Figure 8a: Sample code where no redundant computations are performed. An exchange of arrays "y" and "z" are required for loop 250 to produce the correct answer on each processor.

```
C Assume entire halos of a and b valid before loop 150
C Assume entire halos of x must be valid after 250
      do 150 i=LOCAL_START_1STEP,LOCAL_END_1STEP
         y(i) = a(i) - a(i+1) - a(i-1)
         z(i) = b(i) - b(i+1) - b(i-1)
  150 continue
      do 250 i= LOCAL_START_1STEP,LOCAL_END_1STEP
         x(i) = y(i)*z(i) + y(i+1)*z(i-1)
              + y(i-1)*z(i+1)
  250 continue
EXCHANGE "x" HERE
```

Figure 8b: A version of the same code that uses redundant computation. Since "y" and "z" are computed one step into the halo region, their halo regions are up to date after loop 150. Consequently, the exchanges of "y" and "z" after loop 150 can be eliminated.

Performance optimizations have also been built into SMS I/O operations. By default, all I/O is handled by a single processor. Input data is read by this node and then scattered to the other processors. Similarly, decomposed output data is gathered by a single process and then written. Since atmospheric models typically output forecasts several times during a model run, these operations can significantly affect the overall performance. To improve performance, SMS allows the user to dedicate multiple output processors to gather and output these data asynchronously. This allows compute operations to continue at the same time data are written to disk. The use of multiple output processors has been shown to improve model performance by up to 25 percent (Henderson, et al., 1994).

4. SMS Directives.

This section describes, at a high-level, how SMS directives can be used to address the parallel coding issues identified in Section 2. More information on the use of SMS to parallelize codes is available in the SMS Users Guide (Henderson, et al., 1999). In addition, more detailed information and examples about the SMS directives are provided in the SMS Reference Manual (Govett, et al., 1999). Although directives begin with the characters CSMS$, these will generally be dropped for brevity.

The directives DECLARE_DECOMP, CREATE_DECOMP, and DISTRIBUTE, in combination, enable the programmer to decompose the data among the processors. DECLARE_DECOMP names a decomposition. All other directives refer to this name. For models where memory is allocated statically (i.e. FORTRAN common blocks) the programmer also uses DECLARE_DECOMP to define the processor local sizes of decomposed arrays. For dynamically allocated (such as automatic) arrays, SMS re-declares and re-allocates the arrays to the correct processor-local size at run-time. Typically, the DECLARE_DECOMP directive is placed inside an include file which defines global array sizes so that subroutines throughout the model have access to the decomposition.

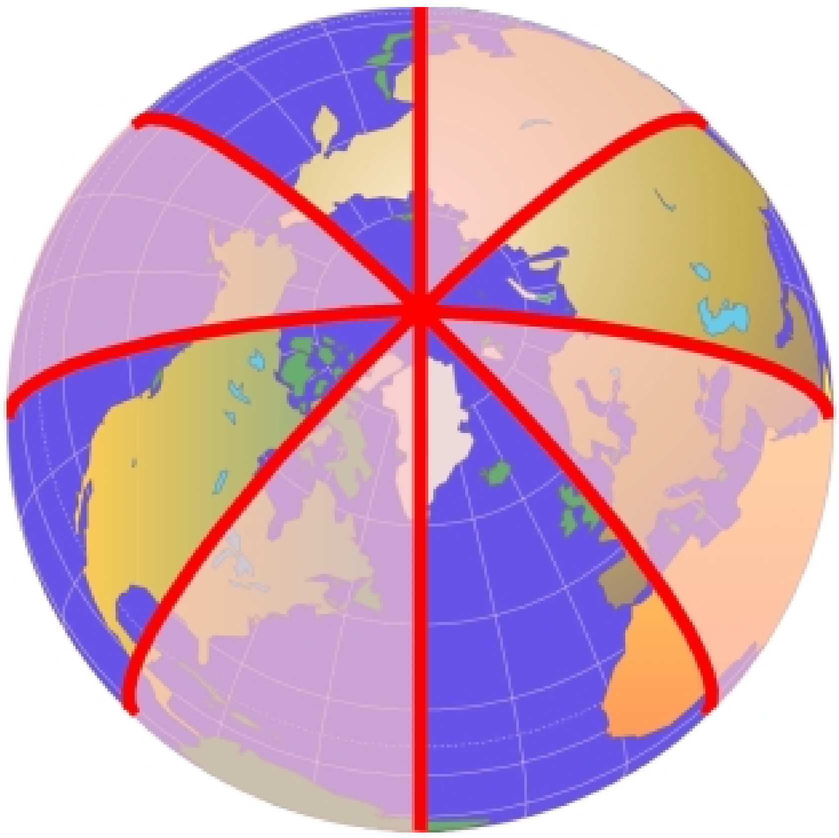


Figure 9: Longitude scrambling is a static load balancing strategy available in SMS. In this example, two processors that are assigned alternating sections of the globe to minimize the load imbalance since shortwave radiation calculations do not occur at “night” points.

CREATE_DECOMP initializes the data decomposition at run-time. It is through this directive that the programmer specifies the number of decomposed dimensions and their global sizes. The sizes of the halo regions are given as well. CREATE_DECOMP also allows the user to address static load imbalances in some situations. For example, if the programmer specifies "SCRAMBLE_LON_STRATEGY" for a decomposed dimension then the data along that dimension is assumed to be along earth meridian lines and is scrambled as shown in Figure 9. As a result, the day-night load imbalance is improved. The DISTRIBUTE directive is used to specify if and how dimensions of individual arrays are decomposed based on the decompositions defined by DECLARE_DECOMP.

Figure 10 shows a simple example of how these directives work together in a static memory model. In this case, there is only one decomposed dimension. The DISTRIBUTE command tells SMS that X and Y are partitioned based on the decomposition named "my_dh". In the translated code, their sizes are statically re-declared to be the size given in the expression "IMWORLD/3+4" specified in the DECLARE_DECOMP directive. The PARALLEL directive in Figure 10 causes the lower and upper bounds of loops 100 and 200 to be translated into processor local equivalents for the decomposition my_dh.

```
integer IM_WORLD
        parameter(IM_WORLD = 15)

CSMS$DECLARE_DECOMP(my_dh, <IM_WORLD/3 + 4>)
CSMS$DISTRIBUTE(my_dh, <IM_WORLD>) BEGIN
        real x(IM_WORLD)
        real y(IM_WORLD)
CSMS$DISTRIBUTE END

C Begin executable code
CSMS$CREATE_DECOMP (my_dh, <IM_WORLD>, <2>)

CSMS$PARALLEL(my_dh, <i>) BEGIN
        do 100 i = 3, 13
          y(i) = x(i) - x(i-1) - x(i+1) -
       &          x(i-2) - x(i+2)
  100   continue
CSMS$EXCHANGE(Y)
        do 200 i=3,13
           x(i) = y(i) + y(i-1) + y(i+1) +
       &           y(i-2) + y(i+2)
  200   continue
CSMS$EXCHANGE(X)
CSMS$PARALLEL END
```

Figure 10: Sample serial code with SMS directives added. SMS directives are the FORTRAN comments that start with the characters CSMS$.

The calculations inside the loops expose an adjacent dependence. Since the arrays X and Y are decomposed, the halo regions must be updated for the code to compute the correct answer. The

EXCHANGE directive generates code that performs the communication needed to do this update. If multiple variables are listed in the directive, the exchanges are aggregated to reduce communications latency. Some exchanges can be eliminated using the HALO-COMP directive. In figure 11, HALO_COMP specifies computations are to extend one step into the halo region in both directions. These redundant computations eliminate the need for an exchange following loop 150.

```
     CSMS$PARALLEL(my_dh,<i>) BEGIN

     CSMS$HALO_COMP(<1,1>) BEGIN
             do 150 i=3,13
               y(i) = a(i) - a(i+1) - a(i-1)
               z(i) = b(i) - b(i+1) - b(i-1)
       150   continue
     CSMS$HALO_COMP END

             do 250 i=3,13
               x(i) = y(i)*z(i) + y(i+1)*z(i-1)
            &       + y(i-1)*z(i+1)
       250   continue

     CSMS$PARALLEL END

     CSMS$EXCHANGE(X)
```

Figure 11: Serial code segment with SMS directives that implement redundant computations in the halo region for loop 150. Exchanges of “y” and “z” are not necessary.

REDUCE is used to address global dependencies deriving from operations such as global sums, maximums and minimums. Figure 12 illustrates a typical reduction operation. In this example, each processor first computes its local sum. The code generated by the REDUCE directive then adds these local sums. Figure 13 illustrates a bit-wise exact reduction. The code between the REDUCE BEGIN and END is replaced with a sum that will insure exactly the same result, regardless of the number of processors.

```
     CSMS$DISTRIBUTE(my_dh, <IM_WORLD>) BEGIN
           Real x(IM_WORLD)
     CSMS$DISTRIBUTE END

     CSMS$PARALLEL(my_dh, i) BEGIN
           xsum = 0.0
           do 100, i=1,15
             xsum = xsum + x(i)
     100  continue

     CSMS$PARALLEL END

     CSMS$REDUCE(xsum, SUM)
```

Figure 12: This example illustrates a global summation operation using SMS directives.

```
      CSMS$DISTRIBUTE(my_dh, <IM_WORLD>) BEGIN
            real x(IM_WORLD)
      CSMS$DISTRIBUTE END

      CSMS$REDUCE(x, xsum, SUM) BEGIN
            xsum = 0.0
            do 100, i=1,15
              xsum = xsum + x(i)
      101   continue
      CSMS$REDUCE(xsum, SUM)
```

Figure 13: This example shows a bit-wise exact global sum of “x” using CSMS$REDUCE.

The TO_GLOBAL and TO_LOCAL directives convert loop indices to their global or local equivalents. In Figure 14, the "i" inside the "max" expression must be treated as a global index so that the expression "i-1" can be properly compared to the global index constant "1". On the other hand, "im1" must be converted to its processor local equivalent because it is used to reference the decomposed variable "y".

```
      CSMS$DISTRIBUTE(my_dh, <IM_WORLD>) BEGIN
            real x(IM_WORLD)
            real y(IM_WORLD)
      CSMS$DISTRIBUTE END

      CSMS$PARALLEL(my_dh, i) BEGIN
            do i=1,IM_WORLD
      CSMS$TO_GLOBAL( <1,i> ) BEGIN
      CSMS$TO_LOCAL( <1,im1> ) BEGIN
              im1 = max(1,i-1)
      CSMS$TO_LOCAL END
      CSMS$TO_GLOBAL END
              x(i) = y(im1)
            enddo
      CSMS$PARALLEL END

      CSMS$REDUCE(xsum, SUM)
```

Figure 14: This example illustrates conversions between global and local address spaces.

Unformatted input and output of decomposed variables is handled automatically, no additional directives are needed. SMS uses information given in the DISTRIBUTE directive to do the appropriate gathering or scattering of the data.

Finally, the TRANSFER directive handles the communication necessary to transform data from one decomposition to another. The source and destination arrays do not need to be decomposed. For example, if the source is decomposed and the destination is not, the TRANSFER directive causes a gather to occur. The converse represents a data extraction where each processor simply copies its local portion of the global array to its local array. (It is not a scatter because each processor already has the global array in memory). Once again, PPP knows from the DISTRIBUTE directives how to generate the correct communication calls. Multiple pairs of arrays can be specified, in which case the transfers are aggregated.

5. Case Studies

The SMS directive-based approach has been used to parallelize three models so far: the atmospheric portion of the U.S. Navy Coupled Ocean/Atmosphere Mesoscale Prediction System (COAMPS) (Hodur, 1997), the FSL quasi-nonhydrostatic (QNH) model (MacDonald, et al., 1999), and the Taiwanese Central Weather Bureau Global Forecast System (GFS) (Liou, et al., 1997). This section focuses on QNH and GFS since measurements of the parallel performance have been made.

The GFS forecast model is a global, atmospheric primitive equation model on sigma coordinates. The forecast model uses a semi-implicit time scheme to integrate five prognostic variables: vertical vorticity, horizontal divergence, potential temperature, specific humidity, and terrain pressure. The horizontal differential of governing equations is calculated by a spectral method that includes fast Fourier transforms and Legendre transforms, while the vertical differential is calculated by an energy-conserving finite differencing scheme. The forecast model includes physical parameterization schemes to model surface fluxes, vertical eddy mixing, shortwave and longwave radiative transfer, cumulus convection, grid-scale condensation, and gravity wave drag. The surface flux parameterization follows Louis (1979) formulas, the vertical eddy mixing parameterization follows Detering and Etling (1985), the radiative transfer parameterization follows Harshvardhan et al. (1987), the cumulus parameterization follows Moorthi and Suarez (1992), the grid-scale condensation calculation simply removes super-saturation level by level, and the gravity wave drag parameterization follows Palmer et al. (1986). The physical processes are calculated on Gaussian grids. The Legendre transform is used to transform the results back to spectral space with no aliasing errors.

Prior to parallelizing the model, it was divided into three programs; a pre-processor, the computational core, and a post-processor. Only the computational core was parallelized since the other pieces are executed just once at the beginning and end of the model run, respectively. The code in the computational core can be, roughly, divided as follows:

1. Physics calculations operating on a latitude-longitude grid.

2. Forward and backward Fast Fourier Transforms operating on latitude-longitude and latitude-zonal wave number grids.

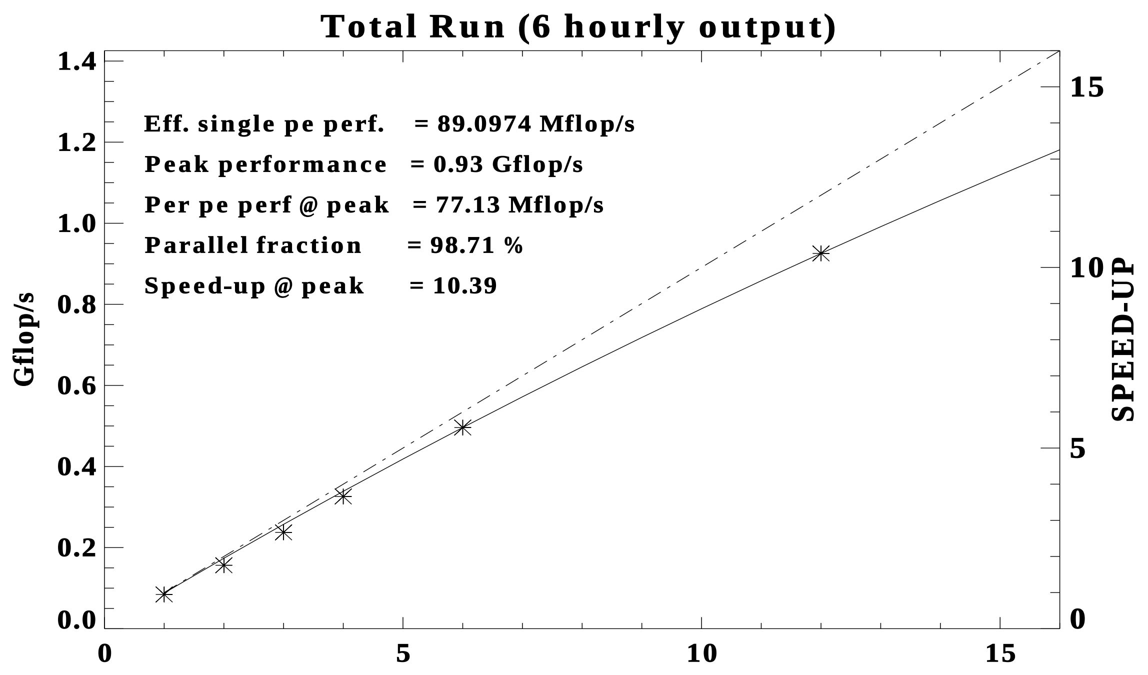
3. Legendre transformations operating on latitude/zonal and zonal/meridional wave number grids.

The target platform for the parallel version of GFS is a Fujitsu VPP 5000 parallel vector processor on-site at CWB. Since this particular machine has a relatively small number of processors (less than 20), it is sufficient to decompose in only one dimension. Two parallel approaches can be taken to handle the Legendre transformations. In the "Global Summation" method the meridional wave number dimension is decomposed and Legendre transformations are computed using global sums. The other approach is to do a process transposition to enable execution of the Legendre calculations over non-decomposed data. These approaches are compared for a variety of resolutions by Foster, et al. (1995). Their study found that the faster method depends on the hardware platform where the code is implemented. As the authors point out, the ideal solution is to support both methods so that the code can run efficiently for any machine type.

Here, the transposition method was chosen since the Legendre calculations will then yield an answer bit-wise identical to the serial code result (regardless of the number of processors), thus simplifying testing. For the physics and FFT calculations, the data are decomposed in latitude. For the Legendre calculations, the data are decomposed in the vertical so that the vector lengths of the inner loops of most calculations are preserved. Scrambling of data was also implemented to alleviate static load imbalances in the radiation and Legendre transformation codes. The radiation imbalances are due to day/night and winter/summer imbalances. For the Legendre transformation code, the imbalances occur because of the way data are stored. The model resolution tested is T120 with 18 levels in the vertical. The code runs to completion on a VPP 300 and produces the correct answer on 1, 2 and 3 processors. A complete analysis of performance on this machine is forthcoming.

A modified version of the code better suited for a cache machine has been developed and tested on a 16 processor 195 Mhz Silicon Graphics Origin 2000 (O2K) platform. In this case, a decomposition in the inner-most dimension was used for the Legendre computations so that cache utilization improves as the number of processors grows. It is important to note that it took an FSL programmer one week to implement and test a decomposition in the longitudinal wave number instead of vertical dimension. The O2K version of the parallel code was tested for 2, 3, 4, 6, and 12 compute processors (plus one I/O processor). Single processor results were measured by compiling the code without translating the PPP directives and then running the code serially.

Model performance can be divided into three phases: initialization, forecast and output. Figure 15a illustrates the parallel performance of the entire model for a 12 hour run with up to 12 processors. These results show that the model scales fairly well. Model performance improves when the effect of initialization, which does not scale, are removed. The operational simulation duration for this model is 4 to 7 days, sufficient to wash out the effects of the start-up serial code time. Thus, the performance of the main time-stepping code (INTGRT) in Figure 15b gives a truer measure of the model scalability. A test was conducted in which data scrambling was turned off. The code was found to run approximately 15% slower, indicating the importance of addressing static load imbalances. This directive-based version of the code shows no loss of efficiency compared to the hand-coded parallel version using only the SMS run-time libraries (results omitted for brevity). These results are preliminary. Further testing is needed on a larger O2K machine as well as other cache-based architectures.



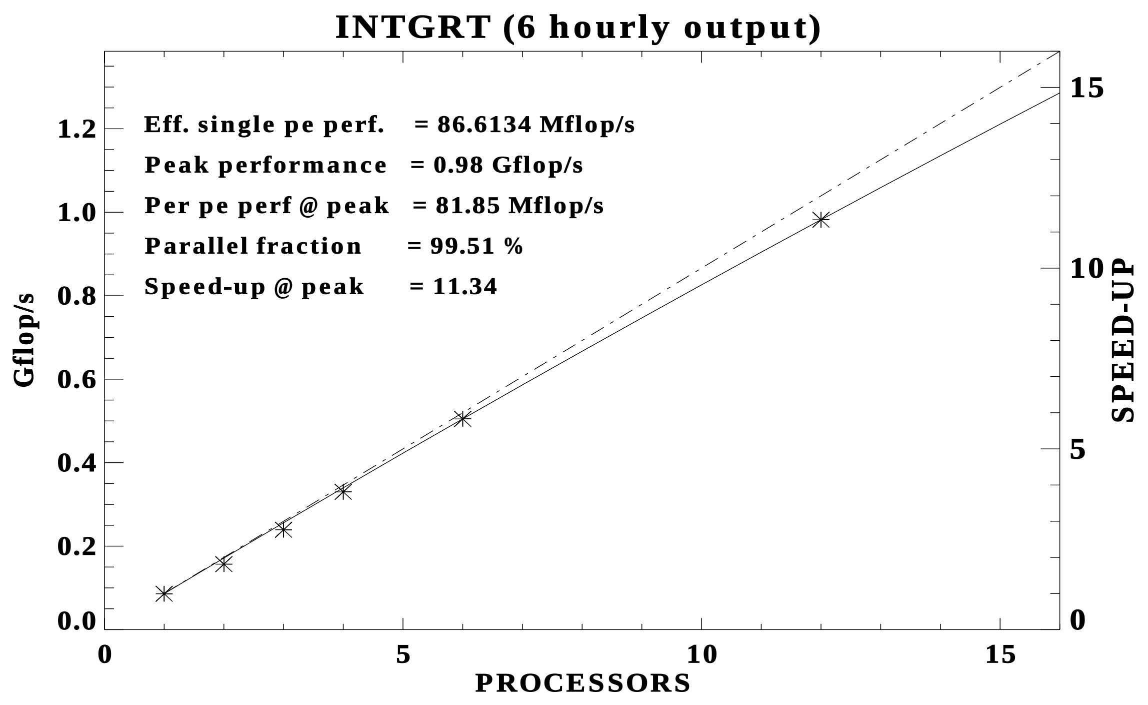


Figure 15: Performance of the SMS parallel version of the GFS model on a 16 processor Origin 2000. The asterisks are measured speeds. The dot-dashed line represents perfect speed-up.

The Quasi-Nonhydrostatic (QNH) model is a high-resolution, grid-based, explicit finite-difference mesoscale limited-area model designed to run on high-performance parallel supercomputers (MacDonald, et al., 1999). The model grid is formulated on the Arakawa C-grid with a terrain-following coordinate in height. The finite-difference approximation for dynamic variables uses a fourth order scheme in space and the third order Adams-Bashforth scheme (Durran, 1991) in time. For the cloud variables a high-order positive definite finite-difference scheme is used (Smolarkiewicz, 1982). The physics packages in QNH includes an explicit cloud physics parameterization suitable for NWP (Schultz, 1995), a broad-band radiation parameterization for mesoscale models based on Pielke (1984), the Mellor-Yamada (1974) turbulent scheme, and a simple vegetation parameterization.

QNH is decomposed in both horizontal dimensions to avoid complex dependencies in the vertical. Except for diagnostic print statements, all horizontal dependencies are adjacent, so only EXCHANGE communications are needed. Redundant computations are implemented to reduce the amount of communications required. The halo thickness is 3.

Figure 16 shows performance on the same Origin 2000 machine described above. The figure shows the model "super-scales", presumably due to the data fitting increasingly well in cache as the number of processors increase. The single processor case performs at 67 Mflops. A complete analysis of these results is forthcoming. The same code has also been run on a set of Intel Pentium III processors connected via Myrinet.

Processors	Time (sec.)	Speedup	Efficiency
1	45858	1.00	1.00
4	9611	4.77	1.19
9	4229	10.84	1.2
10	3897	11.77	1.18

Figure 16: Performance of an SMS parallel version of QNH on a 195 Mhz Origin 2000 platform. Results are for a 42-hour forecast of a 74x74x33, 20 km resolution problem.

Hand-coded SMS versions of finite difference codes such as the Rapid Update Cycle model (RUC) scales well to over one hundred processors. Figure 17 shows the parallel performance of a 40 km version of RUC running on the Intel Paragon. This figure is a tabular form of Figure 8 in Rodriquez, et al. (1996). Since PPP does not add any appreciable overhead to the SMS libraries, it is expected scalability of directive-based versions of QNH, RUC and other FDA codes will be equally good.

Processors	Time (sec.)	Speedup	Efficiency
18	7192	1.00	1.00
28	4698	1.53	0.98
39	3479	2.07	0.95
53	2634	2.73	0.93
68	2093	3.44	0.91
86	1772	4.06	0.85
105	1503	4.79	0.82
127	1291	5.57	0.79
154	1168	6.16	0.72

Figure 17: Performance of a hand-coded SMS parallel version of RUC on the Intel Paragon. Results are for a 12-hour forecast of a 40 km resolution problem over the continental United States.

6. Conclusions and Future Plans

A directive based approach to parallelization (SMS) has been developed that can be used for both shared and distributed memory platforms. The method provides general, high level, comment-based directives that allow retention of the serial code nearly intact. The code is portable to a variety of hardware platforms. Preliminary case studies show this approach can be used to develop parallel code on multiple platforms and achieve good performance. The directive-based approach adds little overhead to parallel versions of the same model hand-coded with the underlying SMS libraries. Since these hand-coded versions have been found to be performance portable, it is expected the same is true of the directive-based solution. However, further testing is needed to verify this.

Several enhancements to SMS are planned. Currently, the directives support positional parameter syntax where the order of the directive parameters must be strictly adhered to. One improvement is to provide support for named parameters. Instead of relying on the position where a field occurs in the directive, named parameters rely on the keyword name to identify the field. This will improve the overall clarity of the SMS directives and relax current syntax restrictions. Since SMS only supports FORTRAN 77 code, a second future enhancement is to support such important FORTRAN 90 features as allocatable arrays, array syntax and modules. Third, although a directive-based parallel version of COAMPS has been developed, the nested grid feature of the model is not supported. SMS will be enhanced to include this support. A fourth planned upgrade to the tool is to have the PPP translator generate OpenMP code to accomplish the parallelization. Further, some state-the-of-art machines consist of clusters of symmetric multiple processors (SMP's). For this architecture, a parallel code that implements tasking "within the box" using OpenMP and message passing "between the boxes" using MPI may be optimal. The PPP translator could be designed to generate both message passing and micro-tasking parallel code. Finally, support for dynamic load balancing will be added.

The SMS approach still requires the modeler to do the dependence analysis by hand. A final future enhancement would be to combine it with the semi-automatic dependency analysis and code generation capabilities of a tool such as CAPTools. The idea would be to have the code generator produce SMS directives instead of parallel code. These directives would then be translated by SMS.

References

Detering, H. W. and D. Etling, 1985: Application of the E-e turbulence model to the atmospheric boundary layer. *Boundary-Layer Meteorology*, **33**, 113-133 (1985).

Durran, D.R., 1991: The third order Adams-Bashforth Method: An attractive alternative to leapfrog time differencing. *Monthly Weather Review*, **119**, 702-720.

Foster, I.T., Toonen, B., and P. Worley, 1995: Performance of Parallel Computers for Spectral Atmospheric Models, ORNL/TM-12986, Oak Ridge National Laboratory, Oak Ridge, TN.

Frumkin, M., Jin, H., and J. Yan, 1998: Implementation of NAS Parallel Benchmarks in High Performance FORTRAN, NAS Technical Report NAS-98-009, NASA Ames Research Center, Moffett Field, CA.

Govett, M., Edwards, J., Hart, L., Henderson, T., and D. Schaffer, 1999: SMS Reference Manual, http://www-ad.fsl.noaa.gov/ac/reference.html.

Gray, R., Heuring, V., Levi, S., Soloane, A., and W. Waite, Eli, 1992: A Flexible Compiler Construction System., *Communications of the ACM,* **35**, 121-131.

Harshvardhan, R. Davies, D. Randall, and T. Corsetti, A fast radiation parameterization for atmospheric circulation models. *Journal of Geophysics Research*, **116**, 1138-1156 (1987).

Hart, L., Henderson, T., and B. Rodriguez, 1995: An MPI Based Scalable Runtime System: I/O Support for a Grid Library, http://www-ad.fsl.noaa.gov/ac/hartLocal/io.html.

Henderson., T., Schaffer, D., Govett, M., and L. Hart, 1999: SMS User's Guide, http://www-ad.fsl.noaa.gov/ac/users.html .

Henderson, T., C. Baillie, S. Benjamin, T. Black, R. Bleck, G. Carr, L. Hart, M. Govett, A. Marroquin, J. Middlecoff, and B. Rodriguez, 1994: Progress Toward Demonstrating Operational Capability of Massively Parallel Processors at Forecast Systems Laboratory, Proceedings of the Sixth ECMWF Workshop on the Use of Parallel Processors in Meteorology, European Centre for Medium Range Weather Forecasts, Reading, England.

Hodur, R. M., 1997: The Naval Research Laboratory's Coupled Ocean/Atmosphere Mesoscale Prediction System (COAMPS). *Monthly Weather Review*, **125**, 1414-1430.

Hwang, K.,1993: *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGraw Hill, Inc, 567-578.

Ierotheou, C.S., Johnson, S.P., Cross, M., and P.F. Leggett, 1996: Computer aided parallelisation tools (CAPTools) - Conceptual Overview and Performance on the Parallelisation of Structured Mesh Codes, *Parallel Computing,* **22**, 163-195.

Kothari, S., and Y. Kim, 1997: Parallel Agent for Atmospheric Models, Proceedings of the Symposium on Regional Weather Prediction on Parallel Computing Environments, pages 287-294.

Liou, C.S., J. Chen, C. Terng, F. Wang, C. Fong, T. Rosmond, H. Kuo, C. Shiao, and M. Cheng, The Second-Generation Global Forecast System at the Central Weather Bureau in Taiwan, *Weather and Forecasting,* ***12****,* pp. 653-663 (1997).

Louis, J. F., 1979: A parametric model of vertical eddy fluxes in the atmosphere. *Boundary-Layer Meteorology*, **17,** 187-202.

MacDonald, A.E., Lee, J-L, and Y. Xie, 1999: On the Use of Quasi-nonhydrostatic Models for Mesoscale Weather Prediction. Accepted by *J. Atmos. Sci*.

Mellor, G.L., and T. Yamada, 1974: A Hierarchy of Turbulence Closure Models for Planetary Boundary Layers*, J. Atmos. Sci*. **31**, 1791-1806.

Michalakes, J., 1994: RSL: A Parallel Runtime System Library for Regular Grid Finite Difference Models Using Multiple Nests, Tech. Rep. ANL/MCS-TM-197, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois.

Michalakes, J., 1997: FLIC: A Translator for Same-Source Parallel Implementation of Regular Grid Applications, Tech. Rep. ANL/MCS-TM-223, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois.

Moorthi, S., and M. J. Suarez, 1992: Relaxed Arakawa- Schubert: A parameterization of moist convection for general circulation models. *Monthly Weather Review*, **120,** 978-1002.

Ngo, T., Snyder, L., and B. Chamberlain, 1997: Portable Performance of Data Parallel Languages, Technical paper presented at Supercomputing 97, San Jose, CA.

Palmer, T. N., Shutts, G., and R. Swinbank, 1986: Alleviation of a systematic westerly bias in general circulation and numerical weather prediction models through an orographic gravity wave drag parameterization. *Quarterly Journal of Royal Meteorology Society*, **112**, 1001-1039.

Pielke, R. A., 1984: Mesoscale Meteorological Modeling. Academic Press, 612 pages.

Rodriguez, B., Hart, L., and T. Henderson, 1996: Parallelizing Operation Weather Forecast Models for Portable and Fast Execution, *Journal of Parallel and Distributed Computing,* **37,**159-170.

Schultz, P., 1995: An explicit cloud physics parameterization for operational numerical weather prediction, *Mon. Wea. Rev.*, **123**, 3331-3343.

Smolarkiewicz, P. K., 1983: A simple positive definite advection scheme with small implicit diffusion. *Mon. Wea. Rev.*,**111**, 479-486.

The Portland Group, 1999: Parallel Fortran for HP Systems, http://www.pgroup.com/presentations/hpcug99/ppframe.htm.